ulm university universität ulm

An Approach for Modeling and Coordinating Process Interactions

Vera Künzle, Sebastian Steinau,
Kevin Andrews, and Manfred Reichert

Ulmer Informatik-Berichte

Nr. 2016-06
September 2016

An Approach for Modeling and Coordinating Process Interactions

Vera Künzle, Sebastian Steinau, Kevin Andrews, and Manfred Reichert

Institute of Databases and Information Systems, Ulm University, Germany
{vera.kuenzle,sebastian.steinau,kevin.andrews,manfred.reichert}@uni-ulm.de

Abstract. In any enterprise, different entities collaborate to achieve common business objectives. The processes used to reach these objectives have relations and, therefore, depend on each other. Their proper coordination within a process-aware information system requires coping with heterogeneous granularity of processes, unclear process relations, and increased process model complexity due to the integration of coordination constraints into process models. This paper presents the concept of coordination processes, which constitute a means to handle the interactions between a multitude of interdependent processes running asynchronously to each other. Particularly, coordination processes leverage the clear identification of process relations, a defined granularity for processes, and the abstraction from details of the individual processes in order to provide a robust framework, enabling proper coordination support for interdependent processes.

1 Motivation

Process-aware information systems (PAISs) support the modeling, execution and monitoring of individual processes. As a particular challenge, work environments increasingly focus on *collaboration* and *interaction*. Generally, business processes cannot be executed entirely in isolation, but their execution depends either implicitly or explicitly on the progress of related processes. In turn, the PAIS is required to be able to properly enact and coordinate process instances of different type. A proper coordination includes the challenge of coordinating multiple process instances of which the exact quantity is unknown at runtime, which may have different kinds of complex relationships, and which may be executed asynchronously to each other without sacrificing flexibility.

Existing approaches dealing with process coordination [2,6,15,18,21,20] have been neglecting both the issues of *process granularity* and the explicit identification of *process relations*. Heterogeneous process granularity impairs a proper coordination, e.g., certain process models may take a very abstract, high-level perspective, whereas others are fine-grained and detailed. Additionally, if process relations are not clearly identified, many coordination scenarios, including the coordination of transitively dependent processes, cannot be supported. Finally, integrating coordination mechanisms into process models increases model complexity and makes models harder to comprehend.

This paper introduces *coordination processes,* a generic concept for coordinating the interactions of different processes. Coordination processes enforce a homogeneous process granularity by *aligning processes with objects*, i.e., an individual process corresponds to a *lifecycle process* of an object describing its progress. Based on this object

alignment, relations between individual processes can be clearly identified and mapped to *semantic relationships.* This allows for a well-defined and controlled process coordination, a simplified modeling of coordination constraints, and the definition of precise operational semantics for enacting coordination processes. Additionally, semantic relationships represent the foundation for a flexible execution of coordinated processes, interfering only when necessary and at certain points in time. This implies that asynchronous execution of processes becomes possible.

The concept of coordination processes can also be used to properly coordinate processes modeled with any paradigm. Due to lack of space, the paper uses the object-aware approach [11,12,13,10,14] for the explanations, a discussion of adapting other process management approaches will be the subject of future publications. Note that the concept of coordination processes also originates in object-aware process management, where *micro processes* serve as lifecycle process. In summary, the contributions of this paper include *semantic relationships*, which constitute a well-defined model for describing the interactions between processes, and *coordination processes*, which is an independent concept for process coordination that is not message- or rule-based, but instead relies on the semantic relationships. Moreover, the loose coupling between coordination processes and the interacting processes allows for easier maintenance of the processes. The coordinated processes may be modeled with any paradigm, provided the requirements of state-based view and data model are satisfied. Finally, coordination processes are executable by design and have precise operational semantics.

The remainder of the paper is organized as follows. Related work is discussed in Section 2. Section 3 introduces basic definitions and the state-based process abstractions. Process relations are captured in a data model, which is explained in Section 4. In Section 5, coordination processes are presented together with an example to illustrate coordination process modeling. In Section 6, the details of semantic relationships are explained. Section 7 briefly discusses evaluation before concluding the paper in Section 8 with a summary and an outlook on future research.

2 Related Work

Regarding the activity-centric paradigm, several approaches enable a specific kind of coordination. In [21], several workflow patterns for coordinating processes are described. This includes patterns to handle multiple instances of activities, taking runtime knowledge into account. The business process architecture approach [6] identifies generic patterns to describe a coordination between processes. Additionally, [6] comprises an analysis of process architectures and identifies several anti-patterns.

BPEL4Chor [1] extends BPEL by adding coordination support for enabling process choreographies. The BPEL4Chor extension incorporates no changes to BPEL itself, which helps the integration between choreographies and orchestrations. iBPM [2,3] enhances BPMN to support coordination of processes by modeling process interactions. iBPM defines a language that allows modeling interactions and includes formal execution semantics. Proclets [18] are lightweight processes with a focus on process interactions as well. Proclets are based on Petri nets, which allows verifying the correctness of the interactions. Interorganizational workflow support [17,19,22] and Business Pro-

cess Protocols [5,4] consider the coordination of business processes between different organizational entities as well.

Common to all these approaches is the use of messages as a mechanism for coordination. While the exchange of messages allows for a detailed process coordination, all message flows have to be identified, the contents of the messages defined, and the recipients determined. This creates a great complexity when facing numerous processes that need to be coordinated, and in many cases, it even impairs the flexible execution of the involved processes. Except Proclets, the modeling of coordination aspects is integrated into the respective process models, increasing their complexity and hence, aggravates their maintainability. Changing a process model then requires additional efforts, since changes may have an impact on other parts of the model, which then have to be adapted as well. A clear *separation of coordination mechanism* on the one hand and process on the other, therefore, could help to reduce the efforts required for changing the model.

Case handling [20] and artifact-centric process management [15] use the Guard-Stage-Milestone (GSM) meta-model [9] for process modeling. Central to them is the *case/artifact*, which holds all relevant information. It may further interact with other cases or artifacts. However, GSM does not provide dedicated coordination mechanisms, but incorporates a sophisticated expression framework, that, in principle, allows creating the needed coordination mechanisms with expressions. As a drawback, these expressions might become very complex and need to be explicitly integrated into the process model. By contrast, in coordination processes, semantic relationships form a basic model to describe process interactions, abstracting from the complexity by defining basic, but customizable, process interaction patterns between processes. In comparison to a purely rule-based approach like GSM, this allows for a simpler and more streamlined modeling of interactions. Coordination aspects of artifact-centric system have been discussed in [8,7].

3 State-based Process Abstraction

Usually, the fact that processes are modeled with different goals in mind, at different levels of granularity, with different modeling philosophies, and with different modeling elements, results in a heterogeneous set of process models. This heterogeneity poses problems when it comes to the coordination of these processes.

Coordination processes provide two mechanisms for tackling these challenges. The first mechanism is the establishment of one defined level of granularity, with which all processes need to comply. In order to realize this defined level of granularity, processes need to be aligned with *object types*. A formal definition of objects is given in Definition 1. In the following, object types are written with capital letters.

Definition 1. *Object type*
Let Identifiers be the domain of all valid identifiers. Let ObjectTypes be the domain of all definable object types. Then $objectType = (name, AttributeTypes, lcProcessType) \in ObjectTypes$ *is a design time representation of a business entity where*

- $name \in Identifiers$ is a unique name.
- $AttributeTypes$ is a set of attribute types.

– *lcProcessType* is a lifecycle process type, describing the progress of $objectType$.

Object alignment is accomplished by each individual process becoming a *lifecycle process* of an object type. This alignment enforces the proper granularity of the processes and further enables the proper identification of process relationships (cf. Section 4 for details). A representative example for lifecycle processes can be found in object-aware process management, where lifecycle process types are called *micro process* types and object types have attributes (cf. [12] for formal definitions).

The second mechanism consists of coordination processes defining a higher-level view on the processes, introducing a layer of abstraction between coordination and lifecycle processes. With regard to process coordination, knowing the specific details of each lifecycle process is, for the most part, not relevant. Sometimes it can even be obstructive to the coordination of processes. The higher level view hides unnecessary details and only leaves the relevant information for coordinating the processes. note that this is similar to the notion of public process views as known from process choreographies [16].

In general, it must be possible to map each individual lifecycle process to a *state-based view*. The state-based view abstracts from details of the lifecycle process and acts as an interface on which the coordination process operates. It allows for a clearer separation of coordination processes and lifecycle processes and an easier modeling of coordination processes. The definition of state-based views is provided in Definition 2.

Definition 2. *State-based view*
A state-based view is a directed, connected graph with $stateBasedView = (process, StateTypeSet, StateTransitionSet)$ *where*

– *process* is the base process for the abstraction.
– *StateTypeSet*, $|StateTypeSet| < \infty$ is a partitioning of *process* into state types.
– $StateTransitionSet \subset StateTypeSet \times StateTypeSet$, $|StateTransitionSet| < \infty$ contains all transitions defined between states.

The elements of the sets are defined as follows:

– $state \in StateTypeSet$ represents a unit of progress within *process* with
  • $state.IncomingTrans \subset StateTransitionSet$ is the set of incoming transitions.
  • $state.OutgoingTrans \subset StateTransitionSet$ is the set of outgoing transitions.
  • $state.Name \in Identifiers$.
– $\exists! startState \in StateTypeSet$ where $|startSate.IncomingTrans| = 0$ is the start state.
– $EndStateSet \subset StateTypeSet$ is the set of end states where $\forall endState \in EndStateSet : |endState.OutgoingTrans| = 0$ holds.
– $trans = (src, tgt) \in StateTransitionSet$ is a transition with $src, tgt \in StateTypeSet$.
– *stateBasedView* must be acyclic.

To obtain a state-based view, the course of execution of a process is partitioned into states. The states are then connected by state transitions according to the order they occur during execution. The state-based view is required to fulfill specific correctness criteria, e.g. acyclicity as stated in Definition 2. In order to illustrate state-based views

and other concepts, this paper uses Example 1 as a running example. It represents a simplified recruitment process from a human resource department.

Example 1. Simplified recruitment process
In the context of recruitment, applicants may apply for job offers. The overall process goal is to determine who of the many applicants is suited best for the job. To evaluate an application, reviews need to be performed. Depending on the concerned department, the number of reviews may differ in order to reject the application or to proceed. Employees of the departments write the reviews and either reject the applicant or suggest inviting him for an interview. In the meantime, more applications may have arrived for which reviews are required, i.e., the evaluation of different applications may be done in parallel. If the majority of reviews are in favor of the application, the applicant is invited for an interview, after which he may be hired or rejected. In particular, when an applicant is hired, all other applicants must be rejected.

Figure 1 shows the generation of a state-based view from the lifecycle process of a *Review* object type from object-aware process management. The lifecycle process is already defined in terms of states. Therefore, all superfluous modeling elements are removed and respective state transitions are put in place. For processes of other paradigms, individual solutions are required, which are outside the scope of this paper.

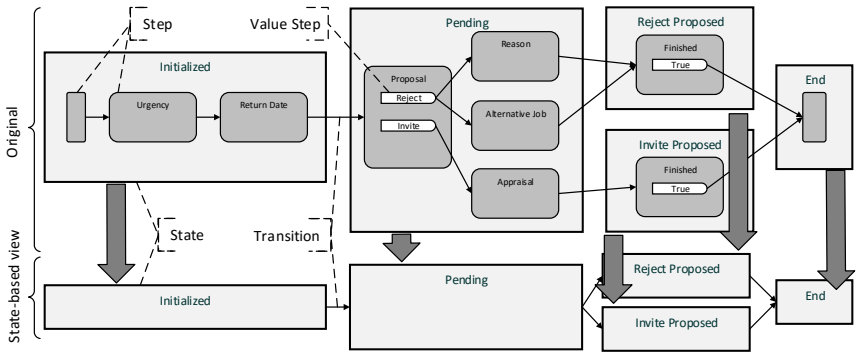

Fig. 1. Creating the state-based view of a *Review* lifecycle process

Furthermore, during the execution of a process it must be ensured that at any point in time exactly one state is active. It is not permitted to have no active state or multiple active states at the same time. Consequently, all outgoing state transitions in the state-based view have an "exclusive OR" split semantics. For example, in Figure 1, states *Reject Proposed* and *Invitation Proposed* are mutually exclusive.

The exact manner in which a state-based view is generated out of a process model is not specified. As long as the result fulfills the requirements (acyclicity, start and end states, state activation), in principle, any process can be abstracted with a state-based view for use with a coordination process. Consequently, the concept of a coordination process can be applied to other process management approaches in the same way. While coordination processes originated from object-aware process management

and are therefore tailored to the lifecycle processes of objects, process models in other paradigms need to be adapted to fit the requirements.

4 Data Model

Coordination processes not only rely on the state-based view, but also on the semantic relationships between processes. Semantic relationships constitute the core concept in coordination processes. To use semantic relationships for coordination, the relations between processes must be identified and explicitly modeled. In general, semantic relationships can be derived from examining the relations between object types. Object types and their relations are semantically captured in a *data model*:

Definition 3. *Data model*
A data model is a tuple $dm = (name, OTypeSet, RTypeSet)$ *representing a directed graph where*

- $name \in Identifiers$ is an identifier.
- $OTypeSet \subset ObjectTypes, |OTypeSet| < \infty$ is a set of object types.
- $RTypeSet \in OTypeSet \times OTypeSet \times \mathbb{N}_0 \times \mathbb{N}_0 \cup \{\infty\}, |RTypeSet| < \infty$ is a set of relations with $relationType = (source, target, min, max) \in RTypeSet$ where
  - $source \in OTypeSet$ is the source object type.
  - $target \in OTypeSet$ is the target object type.
  - $min \in \mathbb{N}_0$ represents the minimum cardinality.
  - $max \in \mathbb{N}_0 \cup \{\infty\}$ with $max \geq min$ is the maximum cardinality.
- dm is acyclic.

Figure 2a shows the data model related to the running example. The model comprises the four object types *Job Offer*, *Application*, *Review*, and *Interview*. Object types, lifecycle process types and relation types are design time entities. At runtime, types can be instantiated to obtain corresponding *instances*. For example, an *Application* type provides the template for creating *Application* instances. Several instances of a specific type may exist at runtime.

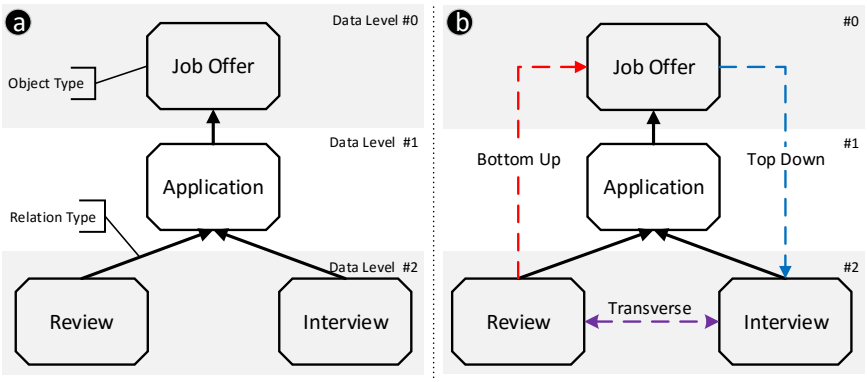

Fig. 2. Data model and semantic relationships

Relation types are displayed as directed edges and represent a 1:n relationship between target and source object type of the relation type. For example, a *Job Offer* instance may have many related *Application* instances at runtime (cf. Figure 2). Each relation type may have assigned cardinalities, restricting the number of object instances at runtime, e.g., an *Application* instance must have at least three related *Review* instances, but not more than seven. Relation types must not create cycles in the data model graph, i.e., cycles must be dissolved to obtain an acyclic data model. The acyclicity of the data model and the directed relations allow for the objects types to be arranged hierarchically (cf. Figure 2), which is required in order to allow for semantic relationships to exist. The hierarchical organization partitions the data model into several *data levels*, formally presented in Definition *4.*

Definition 4. Data levels
Let $dm = (name, OTypeSet, RTypeSet)$ *be a data model.*
Then function $dLevel : OTypeSet \rightarrow \mathbb{N}_0$ *assigns a data level to each* $ot \in OTypeSet$.

$$dLevel(ot) := \begin{cases} 0 & \nexists r \in RTypeSet : r.source = ot \\ 1 + max(\{dLevel(tgt) | & otherwise \\ (ot, tgt, min, max) \in RTypeSet\}) & \end{cases}$$

Object types without outgoing relation types are denoted as *root object types* and are placed on data level #0. It is permissible to have more than one root object type. All other object types are then placed on data levels corresponding to the maximum number of relation types needed to reach a root object type. For example, *Interview* is placed on Data Level #2 as the *path* to the root object type *Job Offer* consists of two relation types. Path determines whether two object types are connected by relation types. Data levels and path are the prerequisites for defining the terms higher-level and lower-level object types, which describe the kind of relation between two object types.

Definition 5. *Path and higher/lower-level object types*
Let $dm = (name, OTypeSet, RTypeSet)$ *be a data model.*
Then: Function $path : OTypeSet \times OTypeSet \rightarrow Bool$ *determines whether a directed path of relations between* $ot_i, ot_j \in OTypeSet$ *exists.*

$$path(ot_i, ot_j) := \begin{cases} true & \exists (ot_i, ot_j, min, max) \in RTypeSet \\ path(ot_k, ot_j) & \exists (ot_i, ot_k, min, max) \in RTypeSet, ot_i \neq ot_k \neq ot_j \\ false & otherwise \end{cases}$$

With path, higher-level object types and lower-level object types can be defined as follows:

$higherLevel : OTypeSet \times OTypeSet \rightarrow Bool$
$higherLevel(ot_i, ot_j) := dLevel(ot_i) < dLevel(ot_j) \wedge path(ot_j, ot_i)$

$lowerLevel : OTypeSet \times OTypeSet \rightarrow Bool$
$lowerLevel(ot_i, ot_j) := dLevel(ot_i) > dLevel(ot_j) \wedge path(ot_i, ot_j)$

The definitions for lower-level and higher-level object types apply analogously also to object instances. The concept of higher-level and lower-level are prerequisites for the definition of the semantic relationship between two object types:

Definition 6. *Semantic Relationships*
Let $dm = (name, OTypeSet, RTypeSet)$ *be a data model and let* $ot_i, ot_k, ot_j \in OTypeSet$ *be object types. Then semantic relationships are defined as follows:*

$$top-down : OTypeSet \times OTypeSet \rightarrow Bool$$
$$top-down(ot_i, ot_j) := higherLevel(ot_i, ot_j)$$

$$bottom-up : OTypeSet \times OTypeSet \rightarrow Bool$$
$$bottom-up(ot_i, ot_j) := lowerLevel(ot_i, ot_j)$$

$$transverse : OTypeSet \times OTypeSet \rightarrow Bool$$
$$transverse(ot_i, ot_j) := \exists ot_k : higherLevel(ot_k, ot_i) \wedge higherLevel(ot_k, ot_j)$$

$$self : OTypeSet \times OTypeSet \rightarrow Bool$$
$$self(ot_i, ot_j) := ot_i = ot_j$$

The terms lower-level and higher-level describe the kind of relation between object types in the data model. A semantic relationship is based on top of these relations and is defined by the way these object types are connected in a coordination processes.

Examples of top-down, bottom-up and transverse relationships are depicted in Figure 2b. The colored edges in Figure 2b, representing the semantic relationships, are depicted solely for illustration purposes, but are not part of the actual data model. Consequently, they are not considered when determining the acyclicity of a data model. An in-depth explanation of semantic relationships and their significance in process coordination is provided in Sections 5 and 6. Note that a coordination process may only coordinate object types having a top-down, bottom-up, transverse or self relationship. It is not possible to asynchronously execute and coordinate object instances at runtime without a semantic relationship.

5 Coordination Processes

Coordination processes are a generic means to coordinate processes by expressing *coordination constraints* with the help of semantic relationships and enforcing them at runtime. A coordination constraint is a statement describing a requirement in regard to process coordination, e.g., an $Application$ must have at least five positive *Reviews* before the applicant may be hired. A coordination process is attached to a particular object type. Object types with an attached coordination process are designated as coordinating object types. It is not allowed to attach more than one coordination process to an object type. Usually, a small data model with no more than five object types has one coordination process, where the coordinating object is also a root object type. In principle however, each object type in a data model may be a coordinating object type to foster the separations of concerns within a large data model.

The fact that the data model is organized hierarchically suggests that a coordination process has a defined area of responsibility in which it may coordinate all object types. The area of responsibility is denoted as the *scope* of a coordination process. In analogy to real-world organizational hierarchies, a coordination process may only coordinate lower-level object types of the object type the coordination process is attached to. The collection of lower-level object types includes the coordinating object type as well. For

example, a coordination process attached to the *Application* object type (cf. Figure 2) is allowed to coordinate *Reviews*, *Interviews* and *Applications*, but is not allowed to coordinate *Job Offers*. With regard to a coordination process coordinating a *Job Offer* and other objects, the coordinating object type must be a higher-level object type of *Job Offer* or the *Job Offer* object type itself.

5.1 Structure of Coordination Processes

Coordination Processes are represented as a directed graph that consists of *steps*, *transitions* and *ports*. Figure 3 shows the coordination process for the running example with *Job Offer* as the coordinating object type. Steps are the vertices of the graph referring to an object type as well as to a state of the object type, e.g. *Job Offer* and state *Published*. For the sake of convenience, a step is addressed with referenced object type and state in the form of *ObjectType*:*State*.

A transition is a directed edge connecting a *source step* with a *target step.* By connecting two steps with a transition, a semantic relationship between the referenced object types is established, e.g. connecting *Job Offer*:*Published* with $Application$:*Sent* constitutes a top-down relationship. note that the sequence in which the steps occur is important for determining the type of semantic relationship. A formal definition can be found in Definition 7.

Definition 7. *Coordination process type*
Let $dm = (name, OTypeSet, RTypeSet)$ *be a data model. A coordination process type* $coProcessType = (coObjectType, PortSet, CoStepTypeSet, CoTransTypeSet)$ *is a directed, connected graph where*

- $coObjectType \in OTypeSet$ is the coordinating object type.
- $PortSet$ is a set of port types with $port = (coStepType, IncomingTransSet)$where
  - $coStepType \in CoStepTypeSet$ is the coordination step type the port is attached to.
  - $IncomingTransSet \subset CoTransTypeSet$ contains all coordination transitions targeting *port*
- $CoStepTypeSet, |CoStepTypeSet| < \infty$ is the set of coordination step types with $coStepType = (objectType, stateType, PortTypeSet) \in CoStepTypeSet$ where
  - $objectType \in ObjectTypeSet$ is an object type related to $coObjectType$ with $path(objectType, coObjectType) = true \vee objectType = coObjectType$.
  - $stateType$ is a state type from the state-based view of the lifecycle process of $objectType$.
  - $PortTypeSet \subset PortSet, |PortTypeSet| < \infty$ is the set of attached port types.
- $CoTransTypeSet \subset CoStepTypeSet \times PortTypeSet, |CoTransTypeSet| < \infty$ is the set of coordination transition types $coTransType = (source, targetPort)$ where
  - $source \in CoStepTypeSet$ is the source coordination step type.
  - $targetPort \in PortSet$ is the target port.
- $coProcessType$ is acyclic.

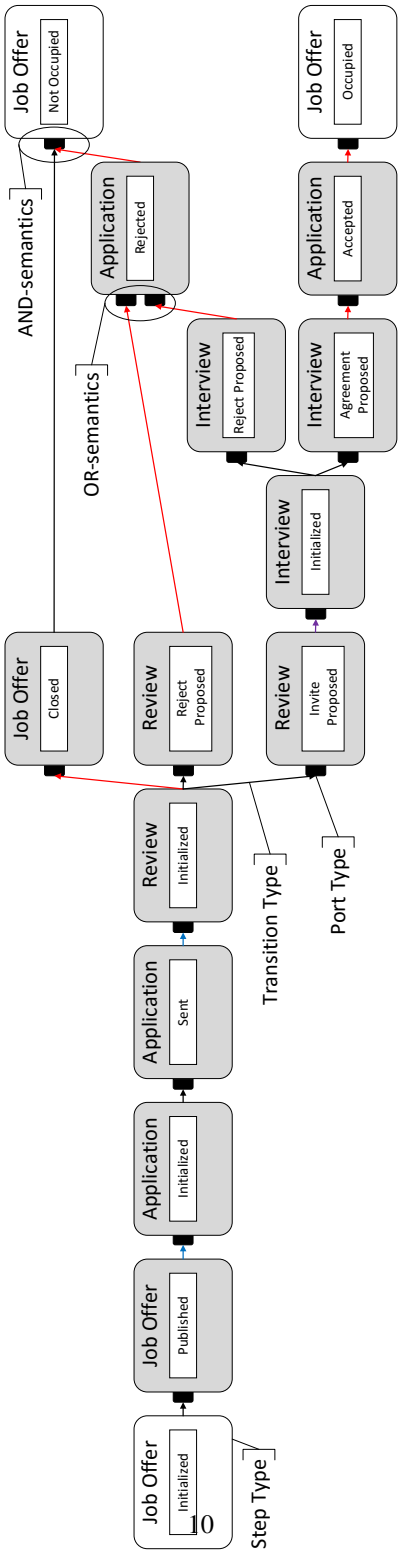

Fig. 3. Complete Coordination Process (Relationships are indicated with color)

For example, assume that a step referencing a *Job Offer* object type is connected to a step referencing the *Application* object type. The source object type is *Job Offer* and the target object type is *Application*, representing a top-down semantic relationship. Reversing the sequence of the steps changes the relationships to bottom-up. The respective relationship is coordinated by a *coordination component*, which is attached to a transition in the graph (cf. Section 6 for details). The coordination components are usually not depicted in a coordination process graph with the aim of keeping the coordination process graph simple. However, macro transitions can be colored to clarify the semantic relationship between source and target without consulting the data model.

During the execution of a process, the progress may not only depend on the fulfillment of a single coordination constraint, but on the fulfillment of multiple coordination constraints at the same time. Similarly, the process may continue when at least one coordination constraint out of many is fulfilled. In order to integrate these *parallel* and *alternative execution paths* within a coordination process, an additional modeling element denoted as *port* exists. A parallel execution path (AND semantics) exists when more than one transition connects to the same port. Conversely, an alternative execution path (OR semantics) exists when multiple transitions connect to different ports attached to the same step. The coordination process in Figure 3 reflects this. Coordination processes can be verified for correctness (e.g. one criterion is that states must appear in the coordination process in the same sequence as in the lifecycle process), even when multiple coordination processes exist in the data model. Coordination processes have to fulfill several correctness criteria. The following selection gives an impression of the challenges the verification of coordination processes has to address.

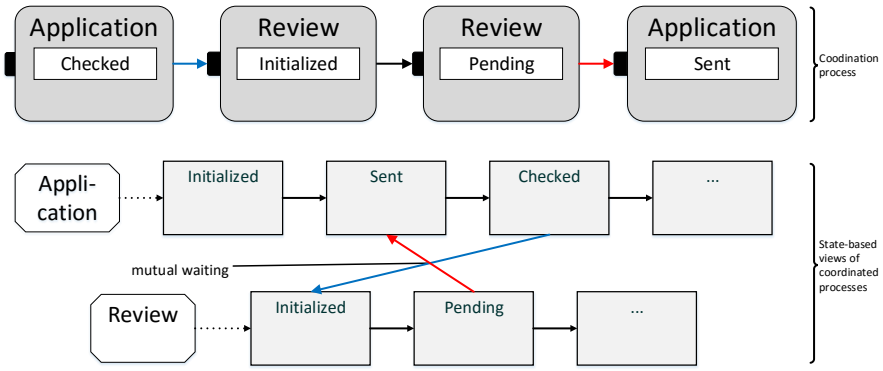

Fig. 4. Example of mutual waiting

A coordination process must have exactly one start step and at least one end step. Any end step must be reachable from the start step. Step that are neither start or end step must also lie on a path from the start step to any end step. In short, the coordination process graph must be weakly connected. In addition to being weakly connected, the

coordination process graph must be acyclic. If the graph contained a cycle, a loop of semantic relationships would be created, which, in turn, implies a loop of coordination constraints. Since the fulfillment of a coordination constraint depends on the fulfillment of the preceding coordination constraints, the loop prevents all coordination constraints from ever being fulfilled. At runtime, if the coordination process progresses to the cycle, the process would be trapped in a permanent deadlock due to *mutual waiting*.

Another case of mutual waiting in a coordination process is depicted in Figure 4. In order for the step *Application*:*Sent* to become active, the coordination process requires that the step which references *Review*:*Pending* is active beforehand. This step however depends transitively on the step *Application*:*Checked*. The lifecycle process defines *Checked* as successor state of *Sent*. Accordingly, the step *Application*:*Checked* can never become active since the activation of state *Sent* is prevented by the coordination process. The mutual waiting results in a deadlock at runtime.

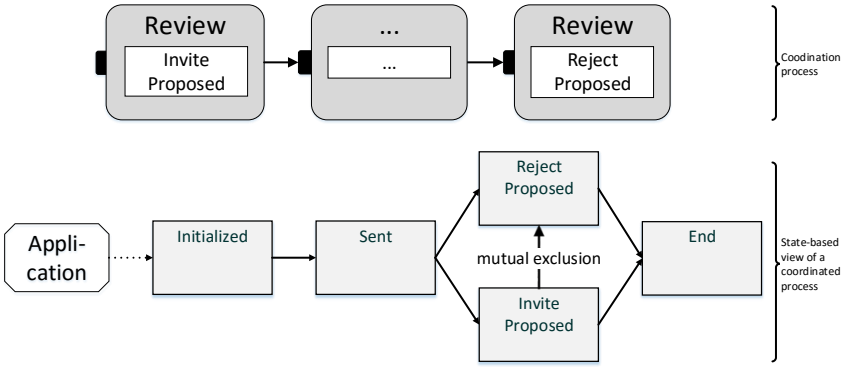

Fig. 5. Example of mutual exclusion

The second source of deadlocks in a coordination process is *mutual exclusion*. In this case, the coordination process has a sequence of steps that references states of alternative execution paths in the lifecycle process. An example of mutual exclusion is presented in Figure 5 using the *Review* object type. The states *Reject Proposed* and *Invite Proposed* can not become active at the same time, therefore the coordination process stops at either step *Review*:*Invite Proposed* or step *Review*:*Reject Proposed*.

Both mutual waiting and mutual exclusion are design time defects and therefore can be detected and resolved at design time. A solution to prevent both problems exists. All steps that refer to the same object type and lie on the same path must refer to a *successor state* of the state referenced in the previous step with the same object type. A successor state of a reference state is a state that can be reached using a path originating from the reference state in the respective lifecycle process graph. The elimination of mutual waitings and exclusions however does not prevent deadlocks from occurring at runtime.

5.2 Coordination Processes at Runtime

At runtime, the object types specified in the data model are instantiated and connected with relations. Overall, this leads to a *complex process structure* of object instances evolving during runtime (cf. Figure 6 for an example). A process structure comprises multiple object instances, their relations, and lifecycle processes. For the sake of simplicity, the lifecycle processes are represented in their state-based views in Figure 6. In general, a process structure may comprise hundreds or thousands of object instances and many more relations.

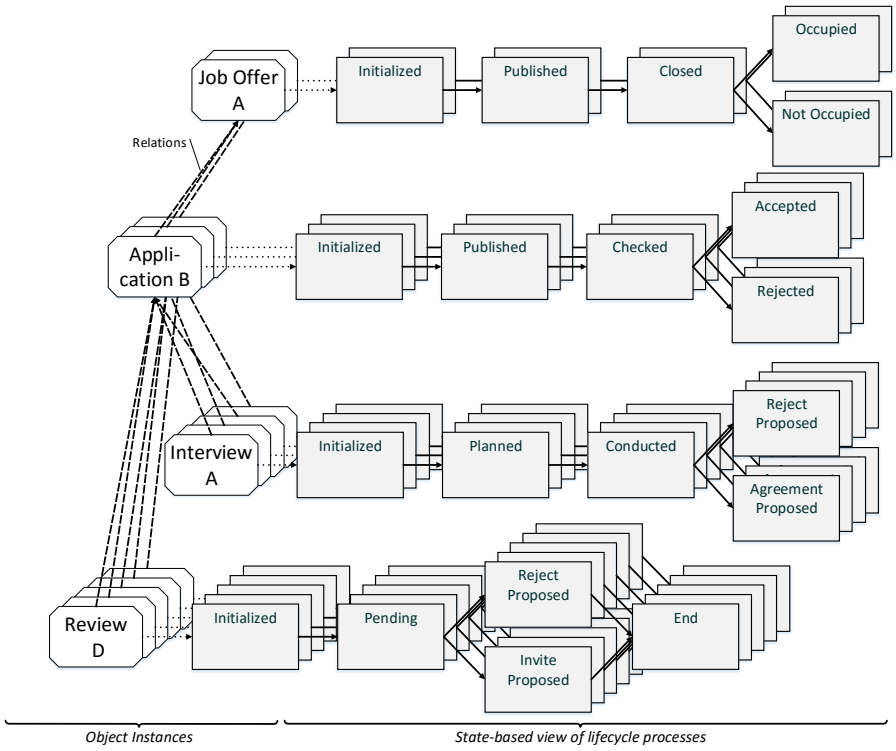

Fig. 6. Example of a complex process structure

As object instances are newly instantiated and others are deleted, the process structure grows and shrinks over time. In addition, relations between these object instances are created and deleted. Finally, object instances may advance their lifecycle processes to a new state. All these dynamic changes must be communicated to the involved coordination processes and handled according to the defined coordination constraints.

The complexity of coordinating a multitude of interrelated objects with their lifecycle processes is abstracted by the coordination process model. In particular, coordination processes are modeled in a flat and compact way, containing only a small number of modeling elements. This is possible due to the structured approach consisting

of data model, state-based view and well-defined semantic relationships. The complex concepts, i.e. relationships and coordination components, can be inferred by examining the steps, ports and transitions in conjunction with the data model. This allows for significantly simplified coordination process modeling without losing functionality or flexibility.

A coordination process coordinates object instances by allowing or prohibiting the activation of lifecycle process states at runtime. When a lifecycle process changes its state, the coordination process is queried for permission. The step referencing the respective process instance and state determines whether or not the state of the lifecycle process may be activated. The step evaluates all attached ports, which, in turn, evaluate the coordination components. Depending on the coordination component and the semantics of the ports (AND/OR), one or more ports may become active. If at least one port is active and, therefore, gives permission, the state of the lifecycle process becomes activated. If no port gives permission, the state is set to pending and must wait until the coordination process can fulfill the coordination constraints.

5.3 Modeling a Coordination Process

To illustrate the modeling of a coordination process and the realization of coordination constraints, the steps involved for creating a part of the coordination process in Figure 3 will be discussed. The example uses the simplified recruitment process involving a *Job Offer*, an *Application* and a *Review* object type. The data model for this example corresponds to the one from Figure 2. The state-based views of the object types involved are depicted in Figures 6. All used coordination constraints are arbitrary and were specifically invented for the example coordination process.

Coordinating object type will be *Job Offer*. A coordinating process begins with its start step, referring to the start state of the coordinating object. For a *Job Offer*, this start state is *Initialized* (cf. Figure 3). Before an applicant may apply for the *Job Offer*, the *Job Offer* must first be published. This constitutes a coordination constraint. To implement the constraint, a new step with reference to *Job Offer* and state *Published* is created. The transition between *Job Offer*:*Initialized* and *Job Offer*:*Published* establishes a self relationship.

Once the *Job Offer* lifecycle process reaches state *Published*, applicants may create *Applications* and apply for the *Job Offer*. Therefore, a new step referring to object type *Application* and state *Initialized* is introduced and connected to the step for *Job Offer* with state *Published* (cf. Figure 3). The created relationship is categorized as top-down, meaning that a *Job Offer* must have reached state *Published* before any *Applications* can be created for the *Job Offer*. Next, the *Application* must be filled out and sent back to the potential employer. In Figure 3, a new Step for an *Application* with state *Sent* is shown, connected to the previous step *Application*:*Initialized*. Since it is the same object in both steps, the relationship is a self relationship, which has no semantics besides allowing an object's lifecycle process to progress through its states normally.

For each *Application* sent to the employer, at least one *Review* must be conducted. Therefore, a step referring to object *Review* and state *Initalized* is added to the coordination process after step *Application*:*Sent* (cf. Figure 3). Like *Job Offer* and *Application*, *Application* and *Review* form a top-down relationship. In turn, the *Review* step with

state *Initialized* enables the activation of several other states. Primarily, an initialized *Review* allows a personnel officer to evaluate the applicant. Thus, advancing the *Review* to state *Reject Proposed* or *Invitation Proposed* becomes possible. Both relationships are categorized as self relationships. When a *Review* becomes initialized, it allows the *Job Offer* to be closed, i.e., no more applicants may apply. The relationship between *Review* and *Job Offer* is categorized as bottom-up. The number of *Reviews* needed to close the *Job Offer* can be specified by the process modeler when configuring the coordination components.

Note that a coordination process does not unnecessarily impede the asynchronous execution of the coordinated processes, e.g. a higher-level process may continue until it needs to wait for the results of lower-level processes. The coordination is only enforced at certain points in time, instead of accompanying all involved processes during their entire execution. This, in turn, allows for a maximum of flexibility when coordinating processes. While the modeling of this example is rather finely grained for the purpose of the demonstration, it is also perfectly acceptable to reduce coordination constraints and allow for a greater flexibility when executing individual processes.

6 Semantic Relationships

The relationships are determined by the connection of steps in the coordination process and the object types they refer to. Figure 7 gives a complete overview over all semantic relationships and the corresponding coordination components. For each semantic relationship a coordination process establishes, a respective coordination component is automatically created.

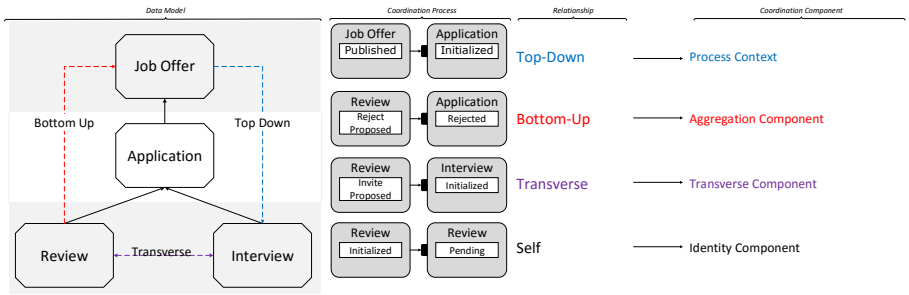

Fig. 7. Relationships and Coordination Components

A *top-down relationship* exists when the execution of lower-level objects depends on the execution of a common higher-level object instance. For example, applicants may only create an *Application* for a *Job Offer* if the *Job Offer* has been published by its creator, i.e. state *Published* is currently active. For each top-down relationship of the coordination process, the coordination process creates a *Process Context Component* that coordinates this specific relationship.

Opposed to top-down relationships, a *bottom-up relationship* means that the execution of a higher-level object depends upon the execution of several lower-level objects. As aforementioned, an *Application* may only be accepted or rejected if a sufficient number of positive or negative *Reviews* exists (cf. Figure 1, states *Reject Proposed* and *Invitation Proposed*). The respective coordination mechanism is the *Aggregation Component.*

In a *transverse relationship*, objects depend on the execution of a set of other objects in context of a common, higher-level object instance. In principle, it is the combination of top-down and bottom-up relationships. According to the data model from Figure 2, an *Application* is a higher-level object type to both *Interview* and *Review*, which have no direct relation to each other. For example, an *Interview* may only be arranged if the respective *Application* has received a sufficient number of favorable *Reviews*. The coordination process monitors the states of all *Reviews* related to the higher-level *Application*. When enough *Reviews* reach state *Invite Proposed*, the component allows the creation of an *Interview* object that belongs to the *Application*. Coordination processes use a *Transverse Component* when a transverse relationship needs to be coordinated.

An *Identity Component* is used for coordinating a *self relationship*. Self relationships are simple and represent the advancement of states in an object's lifecycle process. For example, a *Job Offer* must go from state *Initialized* to state *Published* so an applicant can apply for the job. Therefore, a self relationship is used to obtain state *Published* of the *Job Offer*, which then can be used with a top-down relationship to state *Initialized* of an *Application*.

Coordination components are attached to the transitions of a coordination process. As a result, coordination components have *source* and *target objects* which reside in the transition's source and target step. At runtime, for each higher-level object instance that is referenced in the coordination process, a corresponding coordination component is instantiated. The source and target objects of coordination component only comprise those objects linked to the its higher-level object instance. Because of this, source objects and target objects only comprise a subset of objects referenced by the respective source and target step of the transition. For example, Figure 6 shows a complex process structure with two *Job Offer* instances.

With the establishment of a semantic relationship and the subsequent creation of the coordination component, the role of coordination components in modeling coordination processes is to customize them in order to enable fine-grained control over the respective semantic relationships. In the default configuration, a process context component only becomes enabled when the source object's corresponding state is active, other states deactivate the process context, i.e., no further progress for the target lifecycle processes is allowed. Since all lifecycle processes are executed asynchronously to each other, the higher-level object instance may advance to a subsequent state. This deactivates the coordination component so the target state of the lower-level object instances can no longer be activated.

To keep the process context activated after the source object advances in states, the process modeler can specify a set of states for the higher-level object for which the process context component remains activated. For example, once an *Application* advances from state *Sent* to subsequent state *Checked* (cf. Figure 3), it may still receive

new *Reviews*. This behavior is enabled by adding state *Checked* to the state set of the process context component between steps *Application*:*Sent* and *Review*:*Initialized*. Alternatively, the process modeler may intentionally omit states from the set. For example, state *Closed* is not added to the state set of the coordination component coordinating *Job Offer*:*Published* and *Application*:*Initialized* (cf. Figure 3) to prevent the creation of new *Applications* as soon as the *Job Offer* has been closed.

Aggregation and transverse components use *coordination expressions* to exert control over the underlying relationship. Aggregation and transverse Components permit the activation of their target state when a certain number of source objects are in a particular state. Accordingly, a coordination expression returns a boolean value to indicate whether or not a subsequent state may be activated. For example, an *Application* may only advance to state *Accepted* when at least three related *Interviews* are in state *Agreement Proposed* (cf. Figure 3). Conversely, an *Application* must be rejected when there are more than three *Interviews* in state *Reject Proposed*. The coordination expression representing this coordination constraint is in both cases $\#IN \geq 3$. Cardinalities restrict the number of *Interview* instances at runtime so both coordination constraints can not become true at the same time.

The function #IN counts the number of source objects of the coordination component of which the state specified in the source step is currently activated. All counting functions are context-sensitive. Hence coordination expressions are reusable, as the context of evaluation (i.e. another coordination component) can be changed to obtain different results. A complete list of counting functions required for the proper specification of coordination expressions is presented in Definition 8. In addition, expressions may consist of arithmetic and comparison operators, arithmetic and boolean constants, and boolean operators.

Definition 8. *Coordination Expression Counting Functions*
Let CoordinationComponent be a coordination component with SourceObjectType as the type of the SourceObjects. SourceObjects is the set of objects in the source step of CoordinationComponent. Let StateSet be the set of states of SourceObjectType.

- $\#ALL : SourceObjects \rightarrow \mathbb{N}_0$
  Determines the total number of source objects for the *CoordinationComponent*.
- $\#IN : SourceObjects \times StateSet \rightarrow \mathbb{N}_0$
  Determines the number of source objects of the *CoordinationComponent* where *State* is currently active.
- $\#BEFORE : SourceObjects \times StateSet \rightarrow \mathbb{N}_0$
  Determines the number of source objects of the *CoordinationComponent* where *State* has not yet been active.
- $\#AFTER : SourceObjects \times StateSet \rightarrow \mathbb{N}_0$
  Determines the number of source objects of the *CoordinationComponent* where *State* has been active in the past, but execution has progressed.
- $SKIPPED : SourceObjects \times StateSet \rightarrow \mathbb{N}_0$
  Determines the number of source objects of the *CoordinationComponent* that have progressed to a subsequent state, but *State* has not been activated during execution.

In case no custom coordination expression is specified, aggregation and transverse components default to the expression $\#IN = \#ALL$, meaning the referenced state must be active in all source instances.

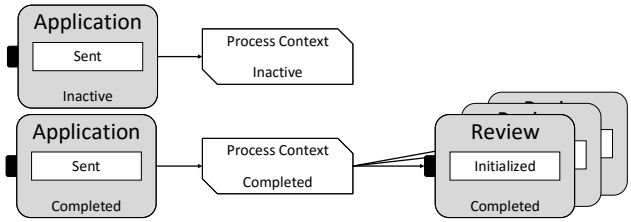

Fig. 8. Runtime example of a process context

At runtime, coordination components monitor the states of the source instances for changes and react appropriately by permitting or prohibiting the activation of the referenced state in the target coordination step. Figure 8 gives an example of process contexts at runtime. The process contexts resides between coordination step *Application*:*Sent* and *Review*:*Initialized* (cf. Figure 3). The semantics of this top-down semantic relationship dictates that *Reviews* may only be created if an *Application* has reached state *Sent*. Currently, two *Applications* are coordinated, each having a dedicated process context. The bottom application already has reached state *Sent*, indicated by the marking *Completed* at the bottom of the coordination step instance. Therefore, the process context allowed the creation of *Reviews*, of which the application has three being in state *Initialized*. The top *Application* is *Inactive* as it has not yet reached state *Sent*. The corresponding process context is also *Inactive* and does not permit the creation of *Reviews* for this *Application*.

In general, semantic relationships provide a structured and consistent foundation for coordinating processes. The complex process structure evolving at runtime can be abstracted using the semantic relationships and is therefore easier to model. Each semantic relationship is managed by a coordination component providing the technical implementation for the semantic relationship. These are customizable, which allows modeling almost any coordination constraint.

7 Evaluation

The concept of coordination processes not only comprises the modeling of process interactions, but includes precise *operational semantics* as well. The operational semantics define the runtime behavior of coordination processes. A business process manage-

ment system prototype based on the object-aware approach has been developed in the Proteus[1] project at Ulm University.

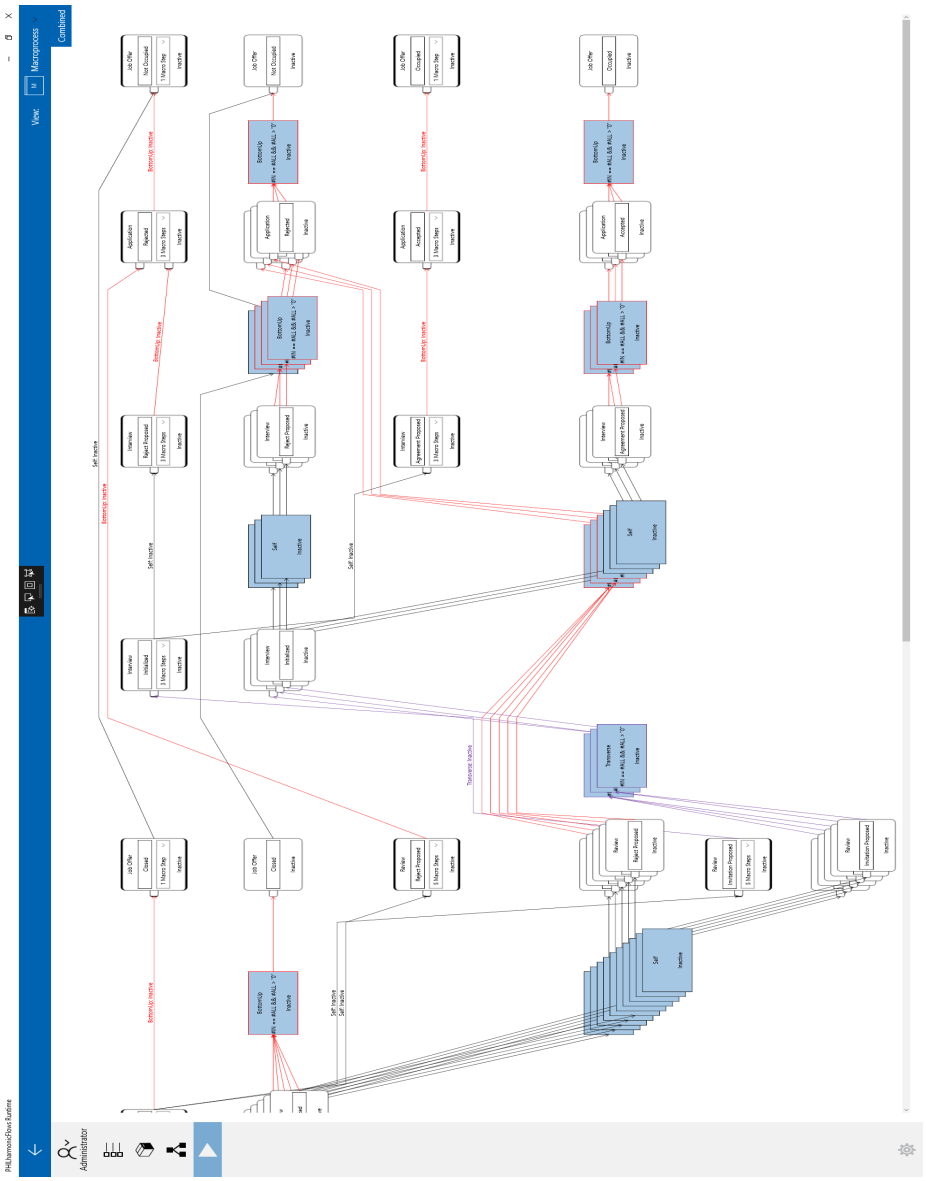

Fig. 9. Screenshot from a coordination process execution in the RDT

[1] For more details on the prototype visit `http://goo.gl/01WwQS`

The prototype comprises a tool for modeling data models with objects together with their lifecycle processes and relations to other processes as well as a runtime environment to enact the modeled processes. Figure 9 shows an execution of a coordination processes in the *Runtime Demonstration Tool* (RDT) as described in this paper. The RDT is the front-end to the runtime environment, which is able to asynchronously execute both lifecycle and coordination processes with the required flexibility. It uses an architecture for high scalability and parallel, asynchronous process execution.

Furthermore, coordination processes were successfully used to model different, real-world processes from Ulm University's administration as well as several processes from human resource departments. Both comprised dozens of object types and multiple coordination processes. The results showed that, in general, coordination processes are able to represent coordination constraints adequately. While modeling of coordination processes has a high learning curve due to different components, it is compensated by the built-in executability of the models and a comparatively easy correctness verification. However, the results also showed several opportunities for improvement, e.g., a support for coordinating different variants of objects was desired.

8 Summary and Outlook

A coordination process is a concept for coordinating a collection of individual processes. The basis for coordination processes is the abstraction of the individual processes using a state-based view. Additionally, a data model is required to make relationships between processes transparent. The coordination process itself is specified in a flat and comprehensive manner using steps, transitions and ports, abstracting from the complexity of coordinating a multitude of interrelated processes. In particular, the coordination processes build on the semantic relationships between processes and use the coordination components Process Context, Aggregation Component, Transverse Component and Identity Component to manage these semantic relationships.

In principle, coordination processes are capable of coordinating processes based on other modeling paradigms, e.g., activity-centric. However, several challenges are still under investigation. These include the search for a standardized way of redesigning processes with heterogeneous granularity of other modeling paradigms and notations to achieve object-alignment. Furthermore, several ways to define state-based view are possible, each with different implications regarding the coordination constraints.

References

1. Decker, G., Kopp, O., Leymann, F., Weske, M.: BPEL4Chor: Extending BPEL for Modeling Choreographies. In: IEEE International Conference on Web Services (ICWS 2007). pp. 296–303 (2007)
2. Decker, G., Barros, A.: Interaction Modeling Using BPMN. In: Business Process Management Workshops: BPM 2007 International Workshops, BPI, BPD, CBP, ProHealth, RefMod, semantics4ws, Brisbane, Australia, September 24, 2007, Revised Selected Papers. pp. 208–219. Springer (2008)
3. Decker, G., Weske, M.: Interaction-centric modeling of process choreographies. Information Systems 36(2), 292–312 (2011)

4. Desai, N., Chopra, A.K., Singh, M.P.: Business Process Adaptations via Protocols. In: 2006 IEEE International Conference on Services Computing (SCC'06). pp. 103–110 (2006)
5. Desai, N., Mallya, A.U., Chopra, A.K., Singh, M.P.: Interaction protocols as design abstractions for business processes. IEEE Transactions on Software Engineering 31(12), 1015–1027 (2005)
6. Eid-Sabbagh, R.H., Dijkman, R., Weske, M.: Business Process Architecture: Use and Correctness. In: Business Process Management: 10th International Conference, BPM 2012, Tallinn, Estonia, September 3-6, 2012. Proceedings. pp. 65–81. Springer (2012)
7. Fahland, D., de Leoni, M., van Dongen, B.F., van der Aalst, W.M.P.: Many-to-Many: Some Observations on Interactions in Artifact Choreographies. In: Proceedings of the 3rd Central-European Workshop on Services and their Composition, ZEUS 2011, Karlsruhe, Germany, February 21–22, 2011. CEUR Workshop Proceedings, vol. 705, pp. 9–15. CEUR-WS.org (2011)
8. Hull, R.: Data-Centricity and Services Interoperation. In: Service-Oriented Computing: 11th International Conference, ICSOC 2013, Berlin, Germany, December 2-5, 2013, Proceedings. pp. 1–8. Springer (2013)
9. Hull, R., Damaggio, E., de Masellis, R., Fournier, F., Gupta, M., Heath,III, Fenno Terry, Hobson, S., Linehan, M., Maradugu, S., Nigam, A., Sukaviriya, P.N., Vaculín, R.: Business Artifacts with Guard-Stage-Milestone Lifecycles: Managing Artifact Interactions with Conditions and Events. In: Proceedings of the 5th ACM International Conference on Distributed Event-based System. pp. 51–62. DEBS '11, ACM, New York, NY, USA (2011)
10. Künzle, V.: Object-Aware Process Management. Ph.D. thesis, University of Ulm (2013)
11. Künzle, V., Reichert, M.: Towards Object-aware Process Management Systems: Issues, Challenges, Benefits. In: Proc. 10th Int'l Workshop on Business Process Modeling, Development, and Support (BPMDS'09). pp. 197–210. Lecture Notes in Business Information Processing, Springer (2009)
12. Künzle, V., Reichert, M.: PHILharmonicFlows: Towards a Framework for Object-aware Process Management. Journal of Software Maintenance and Evolution: Research and Practice 23(4), 205–244 (2011)
13. Künzle, V., Weber, B., Reichert, M.: Object-aware Business Processes: Fundamental Requirements and their Support in Existing Approaches. International Journal of Information System Modeling and Design (IJISMD) 2(2), 19–46 (2011)
14. Marrella, A., Mecella, M., Russo, A., Steinau, S., Andrews, K., Reichert, M.: A Survey on Handling Data in Business Process Models (Discussion Paper). In: 23rd Italian Symposium on Advanced Database Systems (SEBD) (2015)
15. Nigam, A., Caswell, N.S.: Business Artifacts: An Approach to Operational Specification. IBM Systems Journal 42(3), 428–445 (2003)
16. Rinderle, S., Wombacher, A., Reichert, M.: Evolution of Process Choreographies in DYCHOR. In: On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE: OTM Confederated International Conferences, CoopIS, DOA, GADA, and ODBASE 2006, Montpellier, France, October 29 - November 3, 2006. Proceedings, Part I. pp. 273–290. Springer (2006)
17. van der Aalst, W.M.P.: Process-oriented architectures for electronic commerce and interorganizational workflow. Information Systems 24(8), 639–671 (1999)
18. van der Aalst, W.M.P., Barthelmess, P., Ellis, C.A., Wainer, J.: Workflow Modeling using Proclets. In: Cooperative Information Systems: 7th International Conference, CoopIS 2000 Eilat, Israel, September 6-8, 2000. Proceedings. pp. 198–209. Springer (2000)
19. van der Aalst, W.M.P., Weske, M.: The P2P Approach to Interorganizational Workflows. In: Advanced Information Systems Engineering: 13th International Conference, CAiSE 2001 Interlaken, Switzerland, June 4–8, 2001 Proceedings. pp. 140–156. Springer (2001)

20. van der Aalst, W.M.P., Weske, M., Grünbauer, D.: Case handling: a new paradigm for business process support. Data & Knowledge Engineering 53(2), 129–162 (2005)
21. van der Aalst, W., ter Hofstede, A., Kiepuszewski, B., Barros, A.P.: Workflow Patterns. Distributed and Parallel Databases 14(1), 5–51 (2003)
22. Zhao, X., Liu, C., Yang, Y., Sadiq, W.: CorPN: Managing Instance Correspondence in Collaborative Business Processes. Distributed and Parallel Databases 29(4), 309–332 (2011)

Ulmer Informatik-Berichte
ISSN 0939-5091

Herausgeber:
Universität Ulm
Fakultät für Ingenieurwissenschaften, Informatik und Psychologie
89069 Ulm